A brief Tutorial for Implementing Interactive Menus

By Atiyah Elsheikh
a.elsheikh@fz-juelich.de
Institute of Biotechnology
Reseach Center Of Jülich

ABOUT

A brief tutorial about implementing interactive menus specifying various possible configurations for solving a problem. An example of a configuration looks as follows:

```
######                  General    ######
# --help=0                                  # -h : Prints this message
# --stopOnUnknownParam=1                    # Stop if unkown param entered
# --simdir=/home/elsheikh/example           # Path to Modelica Model

######    01. Configuration                     ######
# --optsoft=LEVMAR                          # The optimization software:[LEVMAR|IPOPT|NLOPT|PARADISEO]
--mode=MonteCarlo                           # mode: SingleStart/MultiStart/MonteCarlo REQUIRED
--optimize=0                                # run optimization if conf. complete [0/1] REQUIRED

######    02. Dymosim Specification             ######
# --starttime=0                             # simulation start time [0]
# --stoptime=1                              # simulation stop time [1.0]
# --increment=0.1                           # increment of results : [0.2]
# --algorithm=8                             # DAE Algorithm : [1-14]
# --tolerance=1e-08                         # solver tolerance of results : [1e-8]

######    03. Optimization Specification        ######
# --maxiter=50                              # maximum number of iterations
# --lambda=10                               # start lambda for Levenberg-Marquardt
# --pscaling=none                           # parameter scaling type : [none,log,plog,unity,dynunity]
# --stop_Je=1e-20                           # stop criteria ||J^T e||_{inf} < ?
# --stop_Dp=1e-20                           # stop criteria change in the scaled opt. variables ||Dp|| <
# --stop_e=0                                # stop criteria value of the goal function ||e|| <

######    04. Data                              ######
--datafile=dsin.start.txt                   # input file for generating silico data REQUIRED
# --dseed=0                                 # seed for silico data generation [0]: totaly random
# --dstddiv=0                               # Std Div. for silico data [0]: Exact Data

######    05.c Monte Carlo                      ######
# --nmonts=4                                # No. of Monte Carlo data samples REQUIRED
# --mcseed=0                                # seed for montecarlo data samples, 0: totally random
# --mcstddiv=1e-4                           # Measurements error tolerance
...
```

Here each section represents a menu. Each menu consists of a set of dependent flags, i.e. the existence of a flag depends on the value of another. The existence of a menu may also depend on the value of a certain flag or the consistency of inputs of other menus. The goal is to isolate many information at top-level and not to be hard coded. Additionally, the same main program can be used to solve different configuration of a problem and consistency of information can be examined.

Examples clarified in this tutorial can be found under /home/elsheikh/DymolaSimulator/tutorial. API can be found under /home/elsheikh/DymolaSimulator/docs.

Required Libraries

All required code is under /home/elsheikh/DymolaSimulator/util/ under ibt11 . Additionally, the PARADISEO library (http://paradiseo.gforge.inria.fr/) is used. For compiling the code you will need include

```
$ export DYMSIMHOME=/home/elsheikh/DymolaSimulator
$ export PARADISEOHOME=$DYMSIMHOME/optalg/paradiseo-1.2.1
$ g++ -c -I$(PARADISIEOHOME)/paradiseo-eo/src -I$(DYMSIMHOME)/util main.cpp
```

and link with

```
$ g++ main.o -o mymenus -L$(PARADISIEOHOME)/paradiseo-eo/build/lib -L$(DYMSIMHOME)/util
-lutil -leoutils -leo
```

see Makefile of /home/elsheikh/DymolaSimulator/tutorial

Parser Creation

The first step is to create a parser that creates a text file which includes the desired input specifications. Here how the first program (main.01.cpp) may look like:

```
using namespace std;

#include <eo>
#include "ParserSinglton.h"

/**
 *
 */
int main(int argc, char** argv) {
  // create a parser
  ParserSinglton::create(argc,argv,"current_param.conf");
  eoParser* parser = ParserSinglton::getObject();

  // code for creating flags and menus
  // ...

  // generate the configuration file
  ParserSinglton::generateConfFile();

  // deallocate memory
  ParserSinglton::finalize();
}
```

Compiling the above program

```
$ make
Targets
=======
       build:       build exe without debuging
       clean:       clean all objects
```

```
 $ make build -n
g++ -c -I. -I../util  -I../optalg/paradiseo-1.2.1/paradiseo-eo/src -O  main.cpp
g++ -o mymenus  main.o  -L../optalg/paradiseo-1.2.1/paradiseo-eo/build/lib -L../util -lutil
-leoutils -leo

$ make build
```

The program *mymenus* is generated. Executing this program

```
$./mymenus
```

generates the file *current_param.conf*

```
$ more current_param.conf

######                 General    ######
# --help=0                                  # -h : Prints this message
# --stopOnUnknownParam=1                    # Stop if unkown param entered
```

after editing the above input file and re runing:

```
$./mymenus @current_param.conf
```

current_param.conf get regenerated with the new input fields or new menus required by the new inputs.

Executing with Emacs

Recommendation: use emacs for interactively editing and running mymenus. With emacs you can run commands with *ctrl-shift-!* to regenerate the above input file again after editing / inserting / changing input flags and reopen with *ctrl-f*

Creating Flags

The main units of a configuration file are flags. Flags can be specified as follows (main.02.cpp):

```
#include "ConfigFlag.h"
#include "Condition.h"
...
  // flag specification
  ConfigFlag<int>  flag(eoValueParam<int>(
                          1,                  // default value
                          "flagname",         // long name --flag
                          "flag comments",    // commnets to the flag
                          0,                  // short name
                          false),             // required or not
                        (Condition *) 0);     // Condition to create
  // exploring the flag
  flag.explore(0);

  // accessing flag values
  cout << "flag value of --" << flag.flagKey() << " is " << flag.flagValue() << "\n";
```

For further possible functionalities see API of the classes *FlagInterface* & *ConfigFlag.* Compiling and executing the program produce the file:

```
######                  General    ######
# --help=0                                  # -h : Prints this message
# --stopOnUnknownParam=1                    # Stop if unkown param entered
# --flagname=1                              # flag comments
```

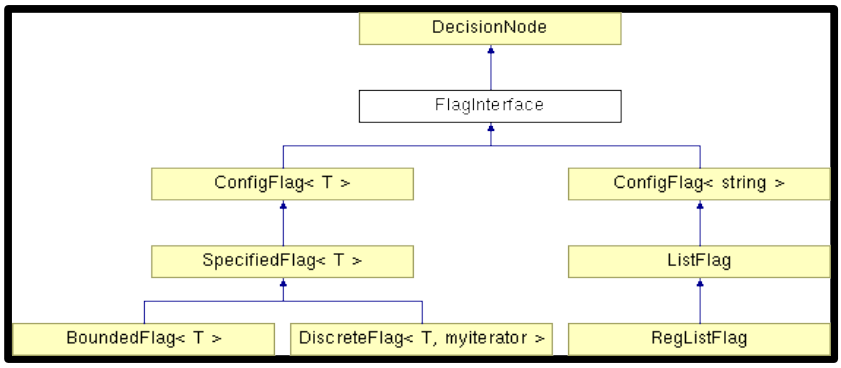
'#' comments the flags. For changing the value of a flag, it should be decommented

```
######                  General    ######
# --help=0                                  # -h : Prints this message
# --stopOnUnknownParam=1                    # Stop if unkown param entered
--flagname=2                                # flag comments
```

otherwise the default value is considered. In the above example the flag is not required, and hence it can be left decommented.

Flag Types

Here is a summary of different type of flags depending on the expected values.



ConfigFlag<T> A simple flag of simple types like int,double,char,string...etc. (--flag=<val_of_type_T>)

SpecifiedFlag<T> Only certain values are allowed

BoundedFlag<T> --flag=<val_of_type_T within_a_certain_range>
(eg. --flag=<val_between_1_10>)

DiscreteFlag<T,Iterator>
like BoundedFlag<T> but the range is specified by an iterator

ListFlag: --flag=str1,str2,str3,str4,... (list of strings like eg. parameter names)

RegListFlag: --flag=regexp1,regexp2,... (list of regular expressions)

Examples

Here are examples declaring *ListFlag* (main.03.cpp)

```
#include “ListFlag.h”
...
  // An example with ListFlag
  ListFlag  listflag(eoValueParam<string>(
                       "",
                     "pars",
                     "parameters to be estimated",
                     0,
                     true),
                    (Condition *) 0);

  listflag.explore(0);

  // accessing the input values
  vector<string> listflagvals;
  listflag.flagValues(listflagvals);

  cout << "flag values of --" << listflag.flagKey() << "\n";
  for(int i=0;i<listflagvals.size();i++)
    cout << "   " << listflagvals[i] << "\n";
```

and DiscreteFlag:

```
#include “DiscreteFlag.h”
...
  // An example with DiscreteFlag
  string Candidates[] = {"Tolga","Michael","Frieder"};

  DiscreteFlag<string,string*>
    inviteMeToIce(                          //iterator of type string*
    eoValueParam<string>(
       Candidates[0],
       "WhoWillInviteMeToday",
       "candidates are: Tolga, Michael, Fireder",
       0,
       true),                               // it is required
    NULL,                                   // without any conditions
    Candidates,                             // iterator begin
    Candidates+sizeof Candidates / sizeof(string));  // end

  inviteMeToIce.explore(0);
  cout << inviteMeToIce.flagValue() << " will invite me to ice\n";
```

A typical input for such flags may look as follows:

```
######                 General    ######
# --help=0                                # -h : Prints this message
# --stopOnUnknownParam=1                  # Stop if unkown param entered
# --flagname=1                            # flag comments
--pars=p1,p2,par3                         # parameters to be estimated REQUIRED
--WhoWillInviteMeToday=Frieder            # candidates are: Tolga, Michael, Fireder REQUIRED
```

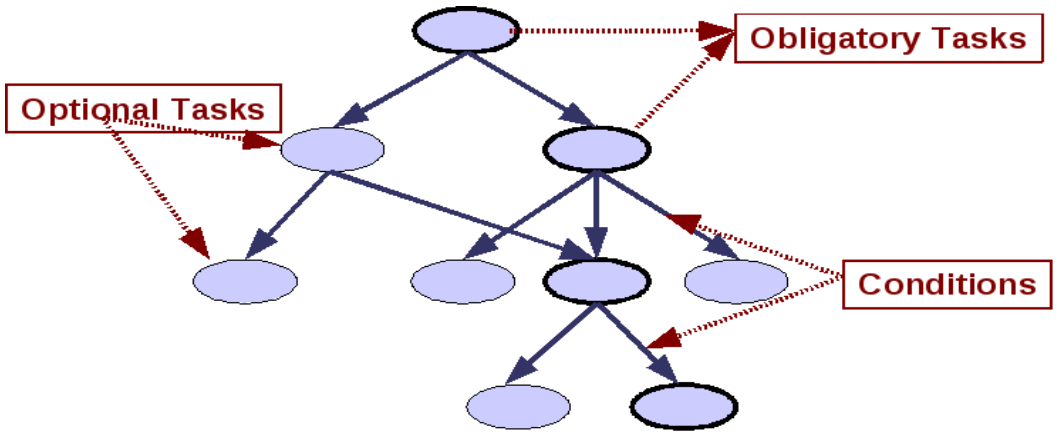
if `–WhoWillInviteMeToday=NoOne` is changed, the program will generate the following outputs:

```
Value of --WhoWillInviteMeToday=NoOne not admissible
```

For creating such flags, see API. Examples how to create and use see DymolaSimulator/menuslib

Decision Trees

Before introducing dependent flags (i.e. flags which values depend on other flags), The concept of decision tree is briefly introduced:

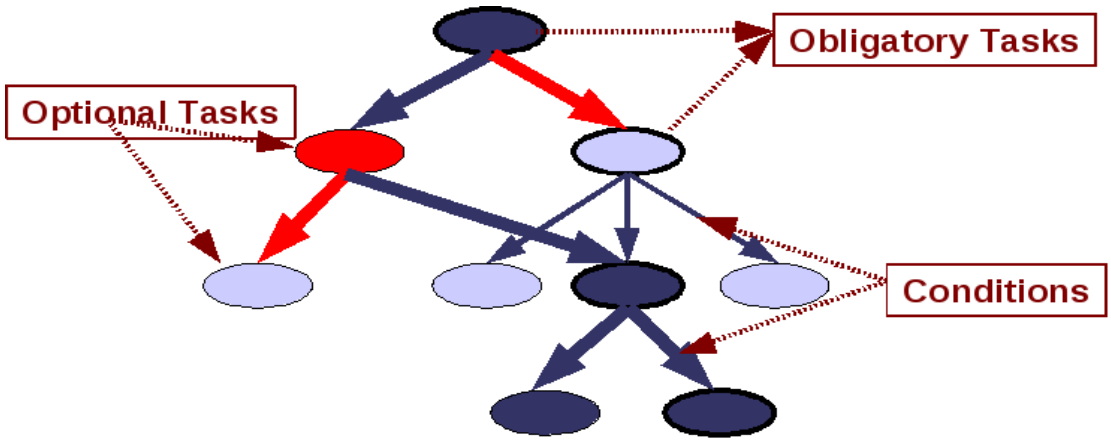


A *Decision Tree* consists of nodes and edges. A *node* represents a task (eg. parsing a flag). A task can be *optional* or *obligatory*. Executing a task can be *successful* or *fail (*eg. when a required flag is not specified or an invalid input is specified by a flag). An *edge* represents a condition for executing the following task.

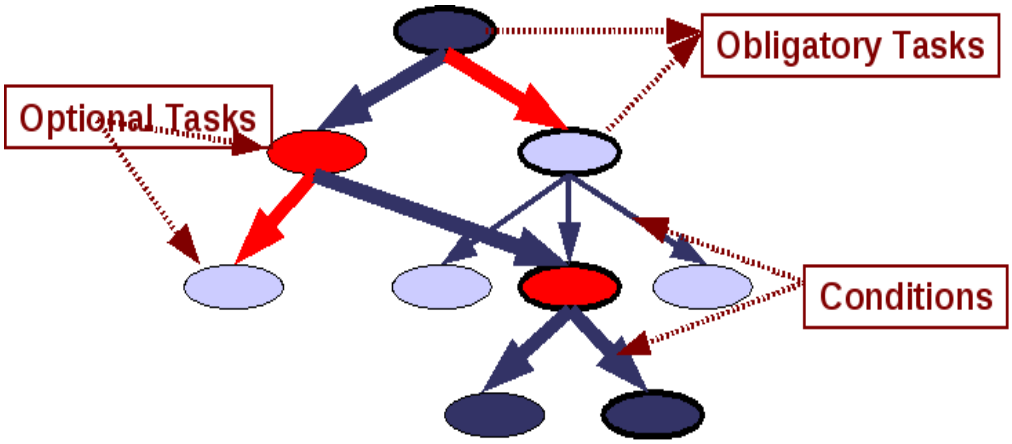
A decision tree is *explored* (i.e. Tasks are performed) by

1. executing the root (i.e. top-level node)
2. examine all outgoing conditions
3. for valid conditions, corresponding tasks are recursively executed according to steps 1,2 and 3

A *successful exploration* occurs when all obligatory tasks which must be performed are successfully executed:



A *fail exploration* occurs if an obligatory task is not successfully performed.



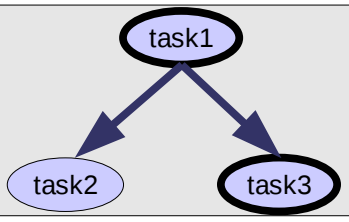
DecisionNode Class

The abstract class *DecisionNode* implements the above concepts. A realization of the above concept can be implemented by extending *DecisionNode* and implementing the protected method *process(void*)* (i.e. a client cannot directly call the method process()). Within this method, the task are determined and performed. The method *process()* is used by the public method *explore()*. A typical extension of *DecisionNode* will look as follows

```
#include "DecisionNode.h"
class MyTask : public DecisionNode {

public:
	MyTask(Condition*,bool optional);
	...
protected:
	bool process(void* data);
	...
}
```

An object of type *MyTask* can perform the following:

```
MyTask task1(cond1,true);
MyTask task2(cond2,false);
MyTask task3(0,true);
task1.insertChild(task2);
task2.insertChild(task3);
bool succ = task1.explore(0);
```



The above code represents 3 tasks. *Task1* depends on *cond1*, *task2* will be performed after performing *task1* if *cond2* is valid. *Task3* is performed after task1 without any condition. The method *explore()* returns whether the exploration was successful or not. For more details, consult the API of *DecisionNode* and the implementation of *ConfigFlag & Menu*.

Dependent Flags

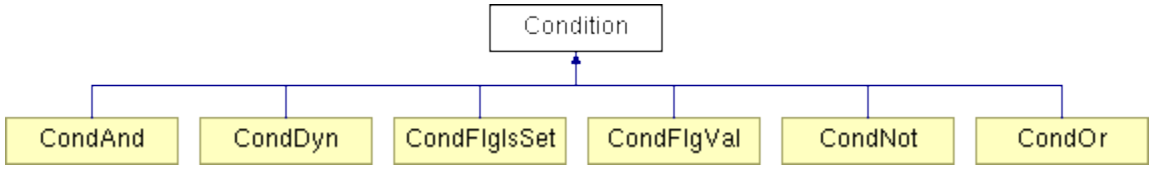
Here is an example clarifying how to implement dependent flags (main.04.cpp):

```
#include "CondFlgVal.h"
#include "CondNot.h"

  // create dependent flags
  ConfigFlag<int>  flag1(eoValueParam<int>(
                          1,                    // default value
                          "flag1",              // long name --flag
                          "flag1 comments",     // commnets to the flag
                          0,                    // short name
                          false),               // required or not
                         (Condition *) 0);      // Condition to create

  // condition expressing whether --flag1=2
  CondFlgVal cond2(&flag1,"2");
  ConfigFlag<double> flag2(eoValueParam<double>(
                            10.0,               // default value
                            "flag2",            // long name --flag
                            "flag2 comments",   // commnets to the flag
                            0,                  // short name
                            false),             // required or not
                           &cond2);             // Condition to create

  // --flag1=any thing except 2
  CondNot cond3(&cond2);
  ConfigFlag<string> flag3(eoValueParam<string>(
                            "ipopt",            // default value
                            "flag3",            // long name --flag
                            "opt software",     // commnets to the flag
                            0,                  // short name
                            false),             // required or not
                           &cond3);             // Condition to create

  // create decision tree of flags
  flag1.insertChild(&flag2);
  flag1.insertChild(&flag3);

  // explore
  bool succ = flag1.explore(0);
```

The above code explores *flag1*. If *–flag1=2*, then *flag2* is explored, otherwise *flag3* is explored.

Condition Types

Several types of Condition exist:



These are summarized as follows:

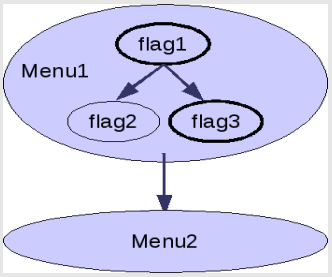
Condition(bool)	Simple condition at static tyme
*CondDyn(bool (*func)())*	Dynamic condition expecting a pointer to boolean function evaluated at run time
CondFlgVal(flag,val)*	whether –flag=val
CondFlgIsSet(flag)*	whether –flag is set by the user
CondNot(cond)*	*the negation of a condition*
CondOr(cond1,cond2)	*whether cond1 or cond2 are valid*
CondAnd(cond1,cond2)	*whether cond1 and cond2 are valid*

Operators like &&, || and ! are not implemented yet. Further types of conditions can be implemented by overwriting the *Condition* class.

Creating Menus

For large set of flags, it makes sense to isolate each related sets of flags into sections called menus. The *Menu* class offers this service. The Menu class implements the class *DecisionNode* and hence it is also possible to implement dependent Menus through decision trees. As follows:

```
MenuType1   menu1(section1,cond1,1);
Flag flag1(..),flag2(..),flag3(..);
flag1.insertChild(flag2);
flag1.insertChild(flag3);
menu1.addFlag(flag1);


MenuType2   menu2(section2,cond2,0);
...
menu1.insertChild(menu2);
```



The Menu class itself is abstract, since it has an abstract class called *specify()*. A *MenuType* class should then extend the Menu class and implements the abstract method *specify()*. In the method *specify()*, it is supposed that related flags would be created and inserted into the menu. It is recommended to isolate Menu in single classes.

Here is an example of a declaration of a menu (MenuLevMar.h)

```
/**
 * \file  MenuLevMar.h
 * \class MenuLevMar provides basic flags for the optimization software LEVMAR
 *
 * @author      Atiyah Elsheikh
 * @date        Mar 2010
 * @since       Mar 2010
 * @version     1.0
 */

#ifndef MENULEVMAR_H
#define MENULEVMAR_H

#include <eo>
#include "Menu.h"
#include "MenuConstants.h"
#include "BoundedFlag.h"
#include "DiscreteFlag.h"

using namespace std;
class MenuLevMar:public Menu {

 protected:
  virtual void specify();

 public:
  /**
   * Cor
   * @param condition to process this node
   * @param optional whether succissful processing
   *        for the menu is necessary for overall success
   */
 MenuLevMar(Condition* cond,bool optional)
   : Menu(MenuConstants::SECOPTM,cond,optional) {
    specify();
  }

  /**
   * Copy Cor
   */
 MenuLevMar(const MenuLevMar& menu)
   : Menu(MenuConstants::SECOPTM,menu._condition,menu.isOptional()) { }

  /**
   * DCor
   */
  virtual ~MenuLevMar() {
    delete _FLG_PSCALING; _FLG_PSCALING = 0;
    ...
  }

  /// candiate values of optsoft flag
  static string CAND_PSCALING[];

  ///
  /// Flags
  ///

  /// flag for optimization software
  DiscreteFlag<string,string*>* _FLG_PSCALING;
  ...

};

#endif
```

The implementation of the above class (MenuLevMar.cpp):

```
/**
 * \file  MenuLevMar.cpp
 * \class MenuLevMar
 */
#ifndef MENULEVMAR_CPP
#define MENULEVMAR_CPP

#include "MenuLevMar.h"
#include "MenuConfiguration.h"

string MenuLevMar::CAND_PSCALING[] =
  {"none","log","plog","unity","dynunity"};


void MenuLevMar::specify() {

  _FLG_PSCALING
    = new DiscreteFlag<string,string*>(
       eoValueParam<string>(
         CAND_PSCALING[0],
         "pscaling",
         "parameter scaling type : [none,log,plog,unity,dynunity]",
         0,
         false),
       (Condition *) 0,
       CAND_PSCALING,
  CAND_PSCALING+sizeof CAND_PSCALING/sizeof(string));

  _FLG_MAXITER = ...

  _FLG_LAMBDA  = ...

  _FLG_STOP_JE = ...

  _FLG_STOP_DP = ...

  _FLG_STOP_E  = ...

  addFlag(_FLG_MAXITER);
  addFlag(_FLG_LAMBDA);
  addFlag(_FLG_PSCALING);
  addFlag(_FLG_STOP_JE);
  addFlag(_FLG_STOP_DP);
  addFlag(_FLG_STOP_E);
}

#endif
```

for a list of other functions of Menus, consult the API.

Menu Types

Several types of Menus, specifically useful for handling large optimization problems, are implemented:

1. Interrelated Variables lists

```
MenuVarsList (ListFlag *inactive, ListFlag *active, ListFlag *cand, const string &section,
Condition *cond, bool optional)
```

Can be used to administrate 3 lists of strings: 1 list is expressing a list of variables to be chosen (eg. Active parameters to be estimated), another list for inactive variables and a list for available candidates. The list of available candidate is provided by implementing the method *allVars()*. This menu ensures that all these lists are mutually exclusive (i.e. no variable appears in two lists). If it is not the case, priority rules are applied to assign each variable only to the list with the highest priority. For more details, see the implementation of /DymolaSimulator/menuslib/MenuDymExpPars.h

2. Interrelated Variables and Regular Expression Lists

```
MenuRegList (RegListFlag *inactive, RegListFlag *active, MenuVarsList *menu, const string
&section, Condition *cond, bool option)
```

is similar to the above, but it deals with regular expressions. In this case it is possible to specify a large set of variables through regular expressions (eg. .*c$: all variables that ends with c). In this case, the corresponding lists of explicit variables are extended by the variables matching these regular expressions. For more details, see the implementation of /DymolaSimulator/menuslib/MenuDymRegPars.h

Here is an example of a possible configuration file:

```
######    06.a Regular Variables                ######
--reginvars=rc                          # list of regular expressions for inactive variables
--regvars=c$                            # list of regular expressions for active variables

######    06.b Explicit Variables               ######
#
--invars=,A.rc_M1.c,A.rc_M2.c,A.rc_M3.c,A.rc_M4.c,A.rc_P1.c,A.rc_P1.r,A.rc_P2.c,A.rc_P3.c,A.
rc_P4.c,A.rc_S1.c,A.rc_S2.c,A.rc_S3.c,A.rc_S4.c,P1.rc_M1.c,P1.rc_M2.c,P1.rc_M3.c,P1.rc_M4.c,
P1.rc_P1.c,P1.rc_P2.c,P1.rc_P3.c,P1.rc_P4.c,P1.rc_P4.r,P1.rc_S1.c,P1.rc_S1.r,P1.rc_S2.c,P1.r
c_S3.c,P1.rc_S4.c,P1ex.rc_M1.c,P1ex.rc_M2.c,P1ex.rc_M3.c,P1ex.rc_M4.c,P1ex.rc_P1.c,P1ex.rc_P
2.c,P1ex.rc_P3.c,P1ex.rc_P4.c,P1ex.rc_P4.r,P1ex.rc_S1.c,P1ex.rc_S2.c,P1ex.rc_S3.c,P1ex.rc_S4
.c,P2.rc_M1.c,P2.rc_M2.c,P2.rc_M3.c,P2.rc_M4.c,P2.rc_P1.c,P2.rc_P1.r,P2.rc_P2.c,P2.rc_P3.c,P
2.rc_P4.c,P2.rc_S1.c,P2.rc_S2.c,P2.rc_S3.c,P2.rc_S4.c,P2.rc_S4.r,P2ex.rc_M1.c,P2ex.rc_M2.c,P
2ex.rc_M3.c,P2ex.rc_M4.c,P2ex.rc_P1.c,P2ex.rc_P1.r,P2ex.rc_P2.c,P2ex.rc_P3.c,P2ex.rc_P4.c,P2
ex.rc_S1.c,P2ex.rc_S2.c,P2ex.rc_S3.c,P2ex.rc_S4.c,S.rc_M1.c,S.rc_M2.c,S.rc_M3.c,S.rc_M4.c,S.
rc_P1.c,S.rc_P2.c,S.rc_P2.r,S.rc_P3.c,S.rc_P4.c,S.rc_S1.c,S.rc_S2.c,S.rc_S3.c,S.rc_S4.c,S.rc
_S4.r,pulse.rc_S1.c,r1.rc_I1.c,r1.rc_P1.c,r1.rc_S1.c,r2.rc_P1.c,r2.rc_S1.c,r2.rc_S1.r,r3.rc_
P1.c,r3.rc_S1.c,r4.rc_P1.c,r4.rc_S1.c,r4.rc_S1.r,r5.rc_P1.c,r5.rc_S1.c # Inactive variables
--vars=A.c,P1.c,P1ex.c,P2.c,P2ex.c,S.c   # Reference Variables, default: all variable
requiring start values REQUIRED
#
--vcand=A.r_net,P1.r_net,P1ex.r_net,P2.r_net,P2ex.r_net,S.r_net,pulse.v,r1.c_I1,r1.c_S1,r1.v
,r2.c_P1,r2.c_S1,r2.v,r3.c_P1,r3.c_S1,r3.v,r4.c_P1,r4.c_S1,r4.v,r5.c_P1,r5.c_S1,r5.v #
Reference Variables Candidates
```

--vcand contains a list of variables that can be chosen by *--vars* (for Specifiying which variables are measured, which measurements would be considered in parameter estimation) or *--invars*. By setting *regvars* and *reginvars*, *vars* and *invars* are automatically generated by matching the regular expressions with the list

available in *–vcand*. The list in *–vcand* are automatically created by the required method *allvars()* by the class *MenuVarList*. It is also possible to manually insert additional variables in an interactive manner using emacs.

3. Extended Lists

```
 MenuExtendedList (ListFlag *list, const vector< string > &postfix, const vector<
ConfigFlag< double > * > &facflags, const string &section, Condition *cond, bool optional)
```

This type of Menus is typically useful for creating flags depending on a specified list. For example list of active parameter bounds). Having set the active parameters (which parameters are active), it is desired to specify bounded box constraints for these parameters. With the help of the above class, we can implement something like:

```
######    10.b Exp Start Values Boundaries / Multi Start     ######
# --r1.k1_min_0=4.5                        #
# --r1.k1_max_0=13.5                       #
# --r2.K_eq_min_0=2.5                      #
# --r2.K_eq_max_0=7.5                      #
# --r2.k1_min_0=0.4                        #
# --r2.k1_max_0=1.2                        #
# --r3.k1_min_0=0.2                        #
# --r3.k1_max_0=0.6                        #
# --r4.K_eq_min_0=1                        #
# --r4.K_eq_max_0=3                        #
# --r4.k1_min_0=0.25                       #
# --r4.k1_max_0=0.75                       #
# --r5.k1_min_0=0.05                       #
# --r5.k1_max_0=0.15                       #
```

Similar Menu type exist for list of regular expressions (eg. For setting generalized boundaries for all parameters that has the name vmax). For an implementation example, consult the class /DymolaSimulator/menuslib/MenuExtListFactory.h

Automatic Generation of code for Flags and Menus

TODO.